DOI: 10.22032/dbt.39948
DOI (proceedings): 10.22032/dbt.39936

vdt Verband Deutscher Tonmeister e.V.

Full Reviewed Paper at ICSA 2019

Presented * by VDT.

A Systematic Performance Investigation of Convolution Algorithms for Synthetic Room Reverberation

S. Heidtmann[1], W. Fohl[2]
[1] *HAW Hamburg, Germany, Email: stefan.heidtmann@haw-hamburg.de*
[2] *HAW Hamburg, Germany, Email: wolfgang.fohl@haw-hamburg.de*

Abstract

This paper presents a systematic investigation of optimization strategies for the convolution algorithm. Special attention is given to features relevant for the creation of virtual room acoustics, where the source signal is convolved with a room impulse response signal which has a length of several seconds. Examined were optimizations for the discrete convolution in the time domain and for the partitioned fast convolution in the frequency domain. Applied technologies were usage of AVX instructions, and GPU computing with the OpenCL framework. The results of the various algorithms are evaluated in terms of sample throughput. Various influence factors on the measured performance were identified. It turned out, that even ambitious projects with more than 10 channels and filter response lengths of several seconds may be rendered in real-time with the GPU version of the discrete convolution.

1. Introduction

Convolution is one of the key algorithms in digital audio processing. FIR filters perform the convolution of the input signal with the filter's impulse response. A special application is the creation of virtual room acoustics in multichannel setups: the original signal is convolved with several directional room impulse responses; the resulting audio signal is played back from the proper directions so that listeners get a realistic impression of the room reverberations.

If the acoustic environment created by this method is supposed to be *interactive*, the virtual room reverberations must be calculated in *real-time*. In order to achieve this goal, several challenges have to be met. Room impulse responses are considerably longer than the FIR filter lengths usually employed in audio processing. Typical reverb durations are in the range of several seconds, resulting in filter lengths of some 100,000 samples.

Approximately 24 reverberation signals will be needed for each primary source to create a realistic room reverb in a WFS system [1].

For interactive environments, *latency* is another important feature. Furthermore, the interaction may cause changes to the virtual room acoustics. These changes must also be recognizable with low latency.

The implementation of the discrete convolution in time domain has an algorithmic complexity of $\mathcal{O}(n^2)$, so for large virtual rooms an optimized implementation of convolution algorithms is of crucial importance. Therefore a systematic investigation of optimization strategies for convolution algorithms has been conducted, taking into account modern techniques as AVX and GPU computing as well as classical code optimization.

The rest of this paper is organized as follows: In the following section related work is given, then the covered optimization strategies are presented. After that the test environment is described and the results are presented and discussed.

2. Related Work

Artificial reverberations is a relatively old topic. The first real-time reverberations where realized in hardware and later by digital filter structures. The real-time usage of convolution reverb is a development over the last 10 years that was enabled by the increasing

power of CPUs and the usage of GPUs for the convolution [2].

That a GPU can be used to increase the performance of different audio processing tasks was shown by L.Savioja, V.Välimäki, J.O. Smith [3]. They showed that with the CUDA framework, a GPU can be used to increase the performance of additive Synthesis, discrete and fast convolution. A problem in performance testing is that the result depends on the hardware. Since the hardware, and the compiler for said hardware, improves with time, the results of older papers may not reflect the current state. Similar results were reported by Wefers and Berg [4].

One of the more recent studies focused on the performance difference between executing the fast convolution on the CPU, showing a performance advantage for the GPU for larger problem sizes [5].

Though most papers focus on the fast convolution, the discrete convolution is of particular interest for real-time audio applications, not only because of their better performance for certain problem sizes. An issue with real-time audio processing is the latency between input and output. The latency depends on the size of the processing buffer, a smaller buffer means a shorter latency. When an effect is added to an instrument in real-time, this latency becomes noticeable. How noticeable depends on the instrument. The acceptable range can range from 1.4 to 42 milliseconds [6]. The processing buffer size for 1.4ms latency would be 62 samples for a sample rate of $44.1\,\mathrm{kHz}$ or 68 samples for $48\,\mathrm{kHz}$.

The properties of the discrete convolution, except the algorithmic complexity, are better for audio processing than the properties of the fast convolution. The size of the processing buffer does not affect the processing time of the discrete convolution, so low latencies are easily achievable, and replacing the filter response takes up to no time. The better performance for smaller problem size makes it potentially a better solution for some problems. Other approaches to the latency problem are hybrid convolution [7] and non-uniformly-partitioned convolution [8]. The Hybrid convolution algorithm only convolves the direct sound and the early reflection and use other methods to create the remaining reverberations. Non-uniformly partitioned convolution convolves the signals for the smaller partitions and calculates the larger ones with the fast convolution.

3. Optimization Strategies

Convolution algorithms have a high algorithmic complexity, but the choice of the algorithm is not the only influencing factor on the performance. A major contributing factor is how well optimized the code is and on what kind of device the code is executed. In this document three approaches to increase the performance of the convolution algorithm by reducing the execution time are evaluated. The first approach is the optimization of the code by standard optimization techniques. The second is the usage of intrinsic functions, functions that call processor specific operations, to improve the performance by using the SIMD unit of a CPU. The last approach is the usage of the GPU of a PC through OpenCL.

Code Optimization Code optimization can be achieved by various means. The options include: reducing the number of CPU operations, replacing slow operation through faster ones, reducing the memory transfer between RAM and CPU Caches as well as between Cashes and the CPU Registers.

Operator Replacing Replacing of slow operation can drastically reduce the execution time of code. Two of the slowest arithmetic operation are the *division* and *modulo* operation, but both can be replaced under certain circumstances.

If many divisions by the same divisor occur, it is more efficient to compute once the inverse of the divisor, and subsequently replace the divisions by the multiplication with the inverse.

Replacing the modulo operation is only possible in integer arithmetic, when both operands are positive and the right operand is a *power of two*. If these conditions are met, the modulo operation can be replaced by a bitwise AND:
$x\%2^n = x\&(2^n-1) | x,n \in \mathbb{N}$

Loop unrolling Loop unrolling is common practice to reduce the number of operations and jumps in the code, since jumps are costly operations. Loop unrolling means to reduce the number of iteration by executing the loop body multiple times in a single iteration. This does not only reduce the number of jumps, but also the number of compare operations [9].

Register Optimization When more variables are used in a code block than the CPU has registers, *register spilling* can occur: The CPU has to store the content of the registers into the cache to free up space for further operations. This slows down the execution time. Usage of the registers can only be directly controlled in assembler but by reducing the number of currently used variables the compiler can optimize the register usage [9].

Advanced Vector Extensions AVX improves the performance of uniform operations on array elements by processing multiple data at the same time. AVX instructions are either used directly in assembler code or by using intrinsic functions in C/C++. AVX can also be used by the compiler if enabled but there is no guarantee that the compiler will use it [9].

OpenCL The Open Computing Language is a framework for developing and executing programs on different platforms for parallel computing, mainly on the GPU, but also on the CPU, on FPGA and DSP [10]. OpenCL differentiates between two components, the *host device* and the *computing device*. The host device acts as a master that starts the execution of OpenCL programs, the so-called *kernels*, and controls the memory transfer between the devices. Each kernel contains a task for the computation of *one single* result element. The kernels are written in Open CL C, a programming language that is based on the syntax of C.

OpenCL code can be optimized by using the same techniques as described previously. To further optimize the performance, the programmer has to properly make use of global and local memory. A common optimization strategy for the memory is *tiling*. In tiling a problem is divided into multiple smaller parts, where each part is small enough that it can be executed in a single working group of cores [11, 12, 13].

4. Convolution Engine Implementations

The discrete and the fast convolution algorithm were implemented multiple times. Each of them multiple times in C++ and OpenCL usually in an unoptimized variant and an optimized variant to show the benefits of the optimizations. While the implementations of the convolution engine interface differ in the used hardware,

```
1  int n = currentRingBufferPos;
2  for (int i = n; i < I/O_Buffer.size; i++, n++){
3   for (int m = 0; m < Filter.size; m++){
4     I/O_Buffer[i] +=
5
        Save_Buffer[(n - m) % Save_Buffer.size] * Filter[m];
6   }
7  }
```

Fig. 1: Pseudocode for the discrete convolution

and the used algorithm, the internal buffer structure is similar. All of them have a buffer for the filter response, the I/O buffer, and a buffer to hold intermediate data.

The convolution algorithms are explained with the help of pseudocode. To improve readability, the pseudocode describes only the convolution of a single channel. The iteration over the different audio channels as well as the normalization of the audio are missing. The normalization is simply a multiplication of all samples in the result and the iteration over the audio channels an additional loop and index for all buffer accesses.

4.1. Discrete Convolution Engine

This engine implements the discrete convolution using the Overlap Save approach. This implementation was created to have an unoptimized implementation as a reference of the discrete convolution for comparison with other engines.

The discrete convolution engine implements the convolution by implementing the equation 1 to calculate a sample for the result.

$$(f*g)[n] = \sum_{m=0}^{|g|} f[n-m] \cdot g[m] \tag{1}$$

The convolution engine implementation needs two `for` loops for processing an I/O buffer (Fig. 1). The first loop iterates over the samples in the I/O buffer and the second loop implements the sum in the equation.

The implementation of the discrete convolution is the implementation of the equation 1 with only a small modification in form of a a modulo operation (Fig. 1, line 4). The modulo operation is necessary because the `Save_Buffer` is a ring buffer. The access to the `Save_Buffer` in line 4 runs backwards. Through the modulo operation the access jumps from the lowest element of the buffer to the highest.

For correct functionality, the ring buffer has to be able to hold at least the same amount of data than the sum of frame size and length of the filter response. This size is necessary because every sample of the filter response is multiplied with a sample in the Save Buffer from a starting point in the reverse direction (Fig. 1). A ring buffer is used in every convolution engine for Overlap Add as well as for Overlap Save.

4.1.1. Optimized Discrete Convolution

Optimizations for reducing the execution time of code are usually applied at the expense of the readability and maintainability of the code (Fig. 2). The most efficient way to optimize code is to optimize the parts of the code that are executed the most, meaning mostly the bodies of loops.

The convolution operation of a single channel in the convolution

```
1  size_t moduloMask = Save_Buffer.size - 1;
2  int n = currentRingBufferPos;
3  for (int i = 0; i < Frame.size; i += 8) {
4    for (int j = 0; j < Filter.length; j += 4) {
5      float filter_0 = Filter[j];
6      // ... etc.
7      float filter_3 = Filter[j + 3];
8
9      float val0 = Save_Buffer[(n + i - j) & moduloMask];
10     // ... etc.
11     float
         val3 = Save_Buffer[(n + i - j - 3) & moduloMask];
12
13     I/O_Buffer[i]     += val0 * filter_0 + val1 * filter_1
14
                          + val2 * filter_2 + val3 * filter_3;
15     // ... etc
16     val1 = Save_Buffer[(n + i - j - 7) & moduloMask];
17     I/O_Buffer[i + 7] += val1 * filter_0 + val2 * filter_1
18
                          + val3 * filter_2 + val0 * filter_3;
19   }
20 }
```

Fig. 2: Pseudocode for the optimized version of the discrete convolution. Shorter and more readable than the actual implementation

```
1  for (int i = 0; i < Frame.size; i += 8){
2    size_t moduloMask = Save_Buffer.size - 1;
3    vecResult = loadValue(0);
4
5    for (int j = 0; j < Filter.size; j++){
6      vecIn = load(&Save_Buffer[(i - j) & moduloMask]);
7      vecFilter = loadValue(Filter[j]);
8      vecResult += vecIn * vecFilter;
9    }
10   store(&I/OBuffer[i], vecResult)
11 }
```

Fig. 3: Pseudocode for the discrete convolution using vector instructions

engine was optimized by *loop unrolling* of the outer loop (Fig. 2, line 3) and the inner loop (Fig. 2, line 4). To avoid register spilling, the number of local variables was reduced by cyclic changing of the `val` variables (Fig. 2, line 9, 11, 16 , etc.).

The last optimization visible in the pseudocode, was to replace the modulo operation with a bitwise AND (Fig. 2, line 9, 11, 16). For this optimization to work, the size of the ring buffer is rounded up to the next power of two.

4.1.2. AVX Discrete Convolution

The vector arithmetic unit of a CPU allows operations on a 256-bit vector. This allows to add or multiply eight floats at the same time. The vectors can be initialized by loading data from a float array (Fig. 3, line 3) and can also be written into a float array (Fig. 3, line 10). The AVX implementation always calculates eight samples for the result at the time. The result is calculated by loading a block of eight samples from the `Save_Buffer` and multiply them with a single value of the filter response. The result of the multiplication is then added to a result register (Fig. 3, line 5 - 7).

The main advantage of AVX is the potentially higher throughput through the use of vector instructions. Additionally, the header of the second loop is executed less often since eight samples are processed at the same time, the same effect as loop unrolling (Fig. 3, line 2).

For clarity, the wrap around of the ring buffer is not shown in the pseudocode.

```
1  copy(transform_time, last_I/O_Buffer)
2  copy(&transform_time[transform_time/2], I/O_Buffer)
3
4  fft(transform_time, transform_freq)
5  copy(fdl[currentPartition], transform_freq)
6
7  int n = currentPartition + nrOfPartition
8  for(int j = 0; j < nrOfPartition; j++)
9    for (int i = 0; i < transform_freq.size; i++){
10     freq_accumulation
         [i] = filter_freq[j][i] * fdl[n - j][i];
11   }
12 }
13
14 ifft(freq_accumulation, transform_time)
15 copy( I/O_Buffer, &transform_time[transform_time/2]);
16
17 currentPartition = (currentPartition + 1) % nrOfPartition
18 last_I/O_Buffer = I/O_Buffer
```

Fig. 4: Pseudocode for the fast convolution with uniform partition

4.2. Uniformly Partitioned Engine

The algorithm with uniformly partitioned filter responses is a variant of the fast convolution that is specificially suited for the block-wise processing in audio applications. This algorithm outperforms in all respects the unpartitioned fast convolution algorithm, where the transformed audio signal is multiplied with the filter response in one single step. So the unpartitioned fast convolution is not discussed further.

The algorithm of the uniformly partitioned engine starts with copying the last I/O buffer into the first half of a transform buffer and the current I/O buffer into the second half (Fig. 4, line 1-2). The transform buffer is then transformed into the frequency domain and its content is put into a FDL (Frequency-domain delay line) (Fig. 4, line 4-5). The convolution is then carried out by multiplying the filter partitions with the entries in the FDL and adding them in an accumulation buffer (Fig. 4, line 7-12). The last step is to transform the accumulation buffer into the frequency domain and copy the second half into the IO buffer (Fig. 4, line 14-15).

A particular optimization is the access to the FDL ring buffer. The required modulo operation could be replaced by an AND operation if the size of the FDL would be increased to a power of two, but instead the size of the FDL is doubled. The first half of the FDL are pointers to the buffers storing the frequency data. The second half is equal to the first half. Since the implementation iterates backward over the entries the implementation starts in the second half. When the wrap around would happen the implementations simply enters the first half.

A version using AVX for the multiplication of the complex numbers exists as well.

4.3. Multithreading

All modern CPU have multiple independent cores. Using them is an effective approach to increase the sample throughput, but additional time is needed for the synchronization of the threads. The multithreading in the convolution engines was designed to minimize the synchronization overhead by assigning the convolution of an audio channel to a single thread. The channels are equally distributed to the threads. Because each thread fully utilizes the computing power of a core, using more threads than there are CPU cores does not increase the sample throughput.

The advantage of this approach is, that the threads are completely independent from the other threads during the convolution. Synchronization is only needed to start the threads and to wait until all threads have completed their task.

```
1  convolve(Frame, Save_Buffer, Filter, n){
2  int channel_id = get_global_id(0);
3  int sample_id = get_global_id(1);
4
5  channel_save = SaveBuffer[channel_id]
6  channel_filter = Filter[channel_id]
7
8   float result;
9   for (int m = 0; m < Frame.Size; m++){
10    result +=
        channel_save[(n + sample_id - m) % channel_save.size]
11            * channel_filter[i];
12  }
13  Frame[channel_id][sample_id] = result;
14 }
```

Fig. 5: Pseudocode for the computation device for the discrete convolution with OpenCL

4.4. OpenCL Implementations

Convolution engines using the discrete and the partitioned convolution have been implemented for Open CL. The OpenCL implementations try to reduce the involvement of the CPU to a minimum. The only task of the CPU is the memory transfer and calling of the kernels. The convolution itself is only carried out on the GPU.

In OpenCL, the optimizations can be categorized into two categories: Optimization of the *kernel code* and optimization of the *memory transfer* between the devices. OpenCL is supported by a range of devices, but the optimizations of OpenCL code were applied to maximize the performance on a GPU. The applied optimization of the code may lead to worse performance on other device types.

4.4.1. Discrete Convolution

The simple implementation of the discrete convolution with the GPU is similar to the implementation of the discrete convolution on the CPU (Fig. 5). Like the CPU version the GPU version implements Overlap Save. The major difference is that the loops of the samples in the I/O buffer is missing. The loop is implemented by spawning a GPU thread for every iteration of the loop. The amount of threads spawned is controlled by the host device (Fig. 6, line 8).

The kernel of all threads is the implementation of the equation for the discrete convolution (eq. 1). The equation calculates a single value for the result. The thread gets the information which channel and sample they have to calculate by the id of the thread. In OpenCL a thread has a three-dimensional id. In this case, the first dimension is the index of the audio channel and the second the sample in the result that the thread has to calculate (Fig. 5, line 2-3). The last dimension is not used.

4.4.2. Kernel Optimization

The main difference between the standard discrete convolution kernel and the optimized version is the usage of *tiling*. Tiling is a technique to improve the sample throughput by dividing a calculation into smaller tiles to make use of the local memory of the device.

In OpenCL, each thread is part of a work group. A work group on the GPU consists of multiple GPU cores for parallel code execution and a shared local memory. The local memory is smaller but faster than the global memory of the GPU. It is comparable to the caches in the CPU and is shared by all threads.

```
cl_cmdQueue.writeBuffer(Frame, FrameCL)

cl_kernel.setArg(0, CL_Frame);
cl_kernel.setArg(1, CL_Save_Buffer);
cl_kernel.setArg(2, CL_Filter);
cl_kernel.setArg(3, currentRingBufferPosition);

cl::NDRange global(Number of Channel, Frame.Size);
cl_cmdQueue.callKernel(convolve, global);

cl_cmdQueue.readBuffer(Frame, FrameCL)
```

Fig. 6: Pseudocode for the host for executing the discrete convolution with OpenCL

The number of cores in a working group varies from one device to another. On GPU it is usually 32 or 64 cores [11, 12, 13].

Because the local memory is limited, larger problems like the convolution have to be divided into multiple tiles. In practice, the processing of the frame buffer is divided into multiple tiles with either 32 or 64 samples each. This is generally like splitting the frame buffer into multiple smaller buffers. Each working group fully processes one tile (Fig. 7) and like the unoptimized version, every thread calculates one sample for the result [11, 12, 13].

For fast transfer between local memory and global memory the loading process of the thread in a working group needs to be aligned (Fig. 7, line 8 - 13). Aligned means that when a thread with the id x transfers element x from local to global memory the thread with id (x + 1) has to do same for the element (x + 1). If the transfer is aligned, the transfer is a single instruction, if not, the GPU needs one instruction for every single transferred value [11, 12, 13].

Read access to the Save Buffer and the filter response is realized in blocks. The threads load a tile sized block of data into the local buffer and then access the local buffer to calculate the result(Fig. 7, line 12, 18, 19). At all times two blocks of the save buffer have to be loaded to correctly calculate the result. The kernel uses a ring buffer to allow this.

4.4.3. Memory Transfer

The convolution can be optimized by a better usage of the involved hardware, mainly the PCI-E bus. The execution time required to process a frame can be divided into three parts (Fig. 9): Data transfer to the OpenCL device, execution of the convolution, and data transfer to the host. This means that during the time the device processes the current frame the memory bus is idling and vice versa.

By changing the host code the available hardware can be better used by implementing a pipeline (Fig. 8). Processing the frame buffer needs three frames. During the first frame the input frame is transferred to the device, in the second frame the buffer is processed, and in the last frame, the processed data is transferred from the device to the host. This means that at any given time three frame buffers are in the pipeline.

The advantages of the pipeline are, that the time for processing a frame depends only on the longest execution time for one of its components and thus increasing the sample throughput. This is only the case when the memory bus is either full duplex or dual simplex, like PCI-E.

There are two disadvantages, namely a latency between in- and output of three full frames and more memory is needed on the computing device. While the latency can be compensated when the size of the I/O buffer could be reduced to a third of the size than otherwise possible, the increased memory consumption can not be circumvented. The cause for the increase in memory consumption is, that the content of a frame buffer is filled by the host while the device needs them for executing kernel. Because of this three frame buffer are needed on the GPU that are switched by the host (Fig. 8, 1-4).

```
convolve(Frame, Save_Buffer, Filter, n,
  local save, local filter, tile_size){
  float result = 0;
  int channel_id = get_global_id(0);
  int tile_id = get_global_id(1);
  int sample_id = get_global_id(2);

  saveIndex = n + sample_id + tile_id * tile_size;
  filterIndex = sample_id;
  bufferPointer = 0;

  save[bufferPointer] = SaveBuffer[channel_id][saveIndex];
  bufferPointer = (bufferPointer + tile_size) % save.size;

  for(int i; = 0; i < Filter.Size / tile.Size; i++){
    saveIndex =
     (saveIndex − tileSize) % SaveBuffer[channel_id].Size;

    save
     [bufferPointer] = SaveBuffer[channel_id][saveIndex];
    filter
     [bufferPointer] = Filter[channel_id][filterIndex];
    bufferPointer
      = (bufferPointer + tile_size) % save.size;

    for (int m = 0; m < tile_Size; m++){
      result
      += save[(bufferPointer + sample_id − m) % tile_size]
              * filter[m];
    }
    filterIndex += tileSize;
  }
  Frame[channel_id
    ][sample_id + tile_id * tile_size] = result;
}
```

Fig. 7: Pseudocode for an optimized version of the discrete convolution with OpenCL. For this code to work all buffer have to have a size that is a multiple of the tile_size

```
temp = outputBuffer;
outputBuffer = processingBuffer;
processingBuffer = inputBuffer;
inputBuffer = outputBuffer;

cl_cmdQueue.writeBuffer(FrameCL[inputBuffer], Frame)
cl_cmdQueue
     .readBuffer(FrameTempOut, FrameCL[outputBuffer])

cl_kernel.setArg(0, FrameCL[processingBuffer});
cl_kernel.setArg(1, CL_Save_Buffer);
cl_kernel.setArg(2, CL_Filter);
cl_kernel.setArg(3, currentRingBufferPosition);

cl::NDRange global(Number of Channel, Frame.Size);
cl_cmdQueue.callKernel(convolve, global);
```

Fig. 8: Pseudocode for the host for executing the discrete convolution with a pipeline aproach for memory transfer and device computation

4.4.4. Uniformly Partitioned Convolution

The OpenCL partitioned convolution implementation uses the CLFFT library for better performance of the Fourier transforms on the GPU. The host code differs from the host code of the discrete convolution. Instead of using a single kernel, three are used to convolve the signal (Fig. 10). One kernel is for the transform, one for the multiplication of the frequencies and one for the inverse

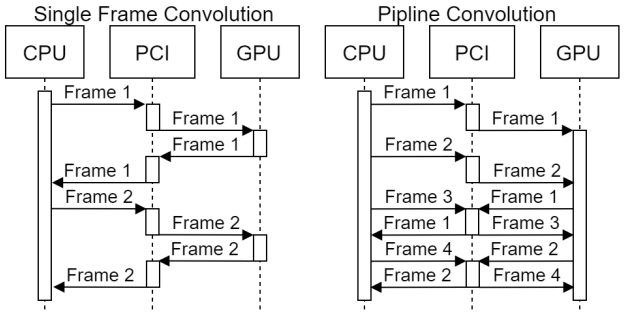

Fig. 9: Pipeline Models for the Open CL Implementation

```
1  copy(transform_time, last_I/O_Buffer)
2  copy(&transform_time[transform_time/2], I/O_Buffer)
3
4  cl_cmdQueue.writeBuffer(transformBuffer, transform_time)
5
6  cl_kernel.setArg(fftargs);
7  cl::NDRange global(Number of Channel, Frame.Size);
8  cl_cmdQueue.callKernel(transform, global);
9
10 cl_cmdQueue.copy(fdl[currentPartition], transformBuffer)
11
12 cl_kernel.setArg(complex multiplication);
13 cl::NDRange global(Number of Channel, Frame.Size);
14 cl_cmdQueue.callKernel(cmplx_mult, global);
15
16 cl_kernel.setArg(ifftargs);
17 cl::NDRange global(Number of Channel, Frame.Size);
18 cl_cmdQueue.callKernel(inverse_transform, global);
19
20 cl_cmdQueue.readBuffer(
       I/O_Buffer, &transformBuffer[transformBuffer.size/2])
21
22 currentPartition = (currentPartition + 1) % nrOfPartition
23 last_I/O_Buffer = I/O_Buffer
24
25
```

Fig. 10: Pseudocode for the host for executing the partitioned convolution

transform.

In the implementation the kernel carries out the convolution by multiplying the filter partitions with the entries, while the FFT and the IFFT are provided by CLFFT. The task of the host is to move the data around, like moving the data from the transform buffer into the FDL (Fig. 10, line 10).

5. Results and Discussion

In this chapter measurement results for the various engines are reported. All measurements were carried out on a Intel Core i7-4770 CPU with 4 cores and a maximum clock frequency of 3.9 GHz and 16 GB RAM. The GPU is a NVIDIA GTX 970 with a maximum clock frequency of 1.316 GHz, 4 GB memory and 13 computing units.

The standard parameter set for the convolution measurements is: I/O *buffer* of 256 samples, *filter length* of 48,000 samples, and 4 *simultaneous channels*.

Only one of these parameters has been varied in the measurements.

5.1. Performance Comparison Between Discrete and Fast Convolution

The plot in Fig. 11 shows the sample throughput of a discrete convolution engine and a fast convolution engine for different processing buffer sizes for all power of two buffer sizes between

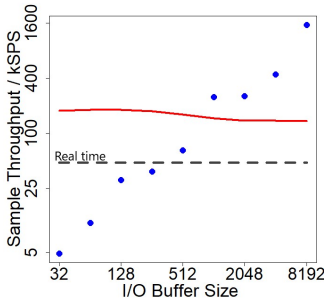

Fig. 11: Sample throughput for the discrete convolution (red) and the fast convolution (blue) for different processing buffer sizes with a filter length of 440,100 Samples. The 48,000 samples per second line is marked by a dashed gray line.

32 and 8192. As seen the in the plot the discrete convolution is mostly unaffected by the change in the buffer size.

On the other hand the sample throughput of the fast convolution grows exponentially with the size of the processing buffer. The discrete convolution has a throughput of roughly 200,000 samples, while the fast convolution starts with 5000 at a buffer size of 32 break the 44,100 mark at a buffer size of 512 and overtake the discrete convolution at a buffer size of 1024. This also means, that unlike the discrete convolution, the convolution of two signals need more time than convolving a single signal with twice the filter length.

The fast convolution data is shown as dots instead of a line in Fig. 11, because the performance of the FFT varies heavily with the length of the transform array. Lengths that are large prime numbers give very poor performance, many small prime factors are good, ideal are powers of two.

5.2. Performance Comparison Between CPU and GPU Implementations

This section presents the results for the most performant implementations of the discrete and partitioned fast convolution, on the CPU and the GPU for various values of I/O buffer size, filter length, and number of channels.

5.2.1. I/O Buffer Size

The first engine parameter tested is the I/O buffer size. While theoretically the size of the I/O buffer does not matter for the performance of the discrete convolution, the GPU performance changes when the I/O buffer size changes (Fig. 12). The lowest sample throughput for the discrete convolution on the GPU is with 40 thousand SPS at an I/O buffer size of 32 samples too low for real-time processing. With increasing buffer size the sample through of this convolution engine increases through to better efficiency of the memory transfer. The sample throughput peaks at a buffer size of 2048 with a throughput of 1524 kSPS. On the CPU the sample throughput is in comparison relatively constant. The peak is at an I/O buffer size of 512 with 175 kSPS and at its lowest at a buffer size of 4096 with a throughput of 136 kSPS. A comparable performance between the two convolution engines is at a buffer size of 128 where the CPU version has with 172 kSPS a slightly higher throughput than the OpenCL version, with 155 kSPS.

For the partitioned convolution the behavior is slightly different (Fig. 12). Like the discrete convolution the OpenCL

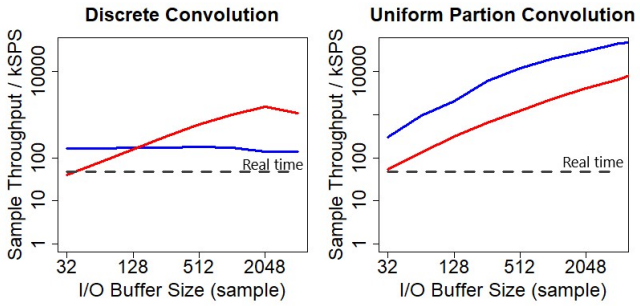

Fig. 12: Sample throughput of the convolution engines by varying size of the I/O buffers. Start value for the buffer size were 32 samples, end 4096 measurement points were all power of twos in between. In this and the subsequent plots, the CPU implementations are in blue, the OpenCL implementations in red, and the 48 kHz threshold for processing audio in real-time for the common sample rates is indicated by the dashed gray line.

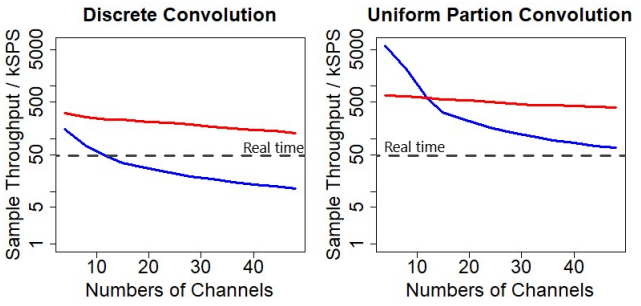

Fig. 13: Sample throughput of the convolution engines by a varying number of channels for the convolution. The start number of channels is 4, and the end 48. Test were executed between start and end in increments of four. Blue: CPU, red: GPU

implementation of the partitioned convolution starts lower than the CPU implementation with a sample throughput of 52 kSPS, but unlike the discrete convolution the OpenCL implementation is not able to surpass the CPU implementation. The CPU implementation starts with a sample throughput of 295 kSPS, a value that is surpassed by the OpenCL implementation at a buffer size of 128 samples. The throughput of both implementations increases with increases in the I/O buffer size. At the last measured buffer size of 4096 the OpenCL implementation has a throughput of 6.6 mSPS, and the CPU version an throughput of 45 mSPS.

5.2.2. Number of Channels

The next parameter to be examined is the number of parallel convolutions. The plots show the behavior of the convolution engines when the number of channels is increased (Fig. 13). The tests start at four channels, are incremented in four channels steps and finally end at a channel number of 48. All convolution engines start with a higher throughput than necessary for a sample rate of 48kHz. No engine with the exception of the CPU discrete convolution fall below this threshold, but the CPU partitioned convolution at channel number 48 is only slightly above the real-time limit with a throughput of 51.5 kSPS. The CPU discrete convolution falls under the 48 kHz threshold when convolving 12 channels. The sample throughput at this point is 46,8 kSPS, less than a third of the sample throughput for four channels 153.5kHz. This sample throughput is too low for a sample rate of 48 kHz but just sufficient for convolving 12 channels at a sample rate of 44.1 kHz.

The sample throughput of the convolution engines decreases with an increase in the number of channels for the OpenCL implementations in a somewhat linear fashion (red line in Fig. 13). On the other hand, the CPU partitioned convolution (blue line) first drastically loses performance and then slows down. The sample throughput of this engine decreases fast from 5.92 mSPS to 585 kSPS at a channel size of 12, the break-even point between the CPU and the OpenCL version.

It is noteworthy, that the performance of the OpenCL partitioned convolution varies only slowly with channel number: The performance at 60 channels is only smaller than the performance at 4 channels by a factor of 1.9.

5.2.3. Filter Length

The last parameter to be examined is the length of the filter. In a first experiment the performance of the discrete and partitioned convolution are measured for rather long filter lengths of 24,000 to 480,000 samples, corresponding to filter durations from 0.5 s to 10 s at 48 kHz sampling rate. The results are shown in Fig. 14.

In a second test, the measurements are repeated for small problem sizes. The results are given in Fig. 15. The goal of these experiments is to find out if there is a break-even point between discrete and partitioned fast convolution. As the plot shows, the uniformly partitioned convolution faster than the discrete convolution even for the smallest filter lengths.

The plots in figure 14 shows that the engine behaves similarly with regard to the filter length as to the number of channels. The OpenCL implementation of the discrete convolution is yet again always better than the CPU implementation, but this time both end up under the 48 kHz threshold. The CPU engine after a filter length of 144,000 samples the OpenCL engine at 288,000 samples, coincidentally the OpenCL discrete convolution engine reaches twice as many samples.

The CPU partitioned convolution outperforms the OpenCL version drastically for shorter filter lengths, but breaks even with it between 144,000 and 168,000 samples. The point when the CPU version falls below the real-time rate of 48,000 samples is reached outside of the scope of the plot at a filter length around 720,000.

The largest filter length that can be processed with our current OpenCL implementation is 5,760,000 samples. At this point, the sample throughput is still 50,000 SPS. This filter length is equal to 120 seconds of audio at a sample rate of 48 kHz.

That the partitioned convolution is not only good for long filter length shows another plot showing the sample throughput for smaller filter lengths for the CPU implementations (Fig. 15). As always the case the partitioned convolution outperforms the discrete convolution. The value range of the discrete convolution is between 25,000 kSPS and 322 kSPS, while the value of the partitioned convolution ranges from 7250 kSPS to 1686 kSPS. This clearly shows that the uniform partition also handles relatively small filter lengths far better than the discrete convolution.

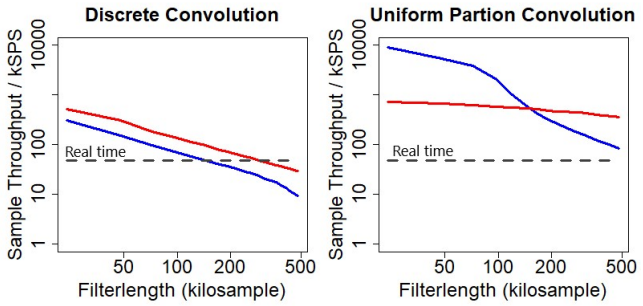

Fig. 14: Sample throughput of the convolution engines by varying filter length. The start filter length is 24,000, and the end 480,000. Tests were executed between start and end in increments of 24,000. Blue: CPU, red: GPU

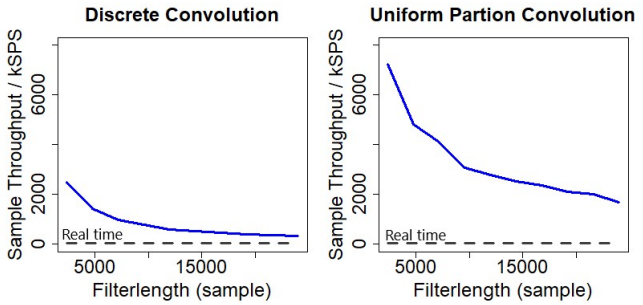

Fig. 15: Sample throughput of the CPU convolution engines for short filter lengths. The start filter length is 2400, and the end 20,600. Tests were executed between start and end in increments of 2400. The size of the I/O buffers is in this case 64.

6. Conclusion

Optimization experiments have been conducted to measure the performance of various convolution implementations. The base parameters were chosen for a typical scenario of virtual room acoustics rendering at 48 kHz: I/O buffer of 256 samples, 4 simultaneous channels, filterlength of 48 kSamples (corresponding to 1 second of room reverb). From this base setting one of the three parameters was varied and the influence on performance was observed. Compared were optimized CPU and GPU implementations in time and frequency domain (discrete convolution and uniformly partitioned fast convolution).

In all experiments, the performance of the frequency-domain implementation was considerably higher, even for small I/O buffers (32 samples), or short filterlengths (2400).
The result of the CPU / GPU comparison is, that the overhead of the GPU implementations in time and frequency domain only pay off at larger parameter sizes. There is one remarkable exception: The CPU-based frequency-domain convolution with 4 channels and 480 kSamples filterlengths is considerably faster at all tested I/O buffer sizes (32 to 4096).

There are however situations in interactive applications, where it is necessary to modify the filter response during runtime. These cases are more easily realized using the discrete convolution. Our results show, that a properly optimized GPU-based *discrete* convolution algorithm is able to handle filterlengths of about 100 kSamples and an I/O buffer size of 256 for some dozens of channels in realtime.

As a proof of concept a VST plugin was developed which allows the selection of the various convolution algorithms and filter responses. With this plugin it was possible to render the acoustics of the WDR concert hall [1] with 24 channels at 48 kHz sampling rate in real-time for our 208-channel WFS system.

7. References

[1] Wolfgang Fohl and Eva Wilk. Enhancements to a wave field synthesis system to create an interactive immersive audio environment. In Proc. 3rd Int. Conf. on Spatial Audio. VDT, 2015

[2] V. Välimäki, J. D. Parker, L. Savioja, J. O. Smith, and J. S. Abel. Fifty years of artificial reverberation. IEEE Transactions on Audio, Speech, and Language Processing **20** (2012), 1421–1448. ISSN 1558-7916

[3] Lauri Savioja, Vesa Välimäki, and Julius O. Smith. Audio signal processing using graphics processing units. J. Audio Eng. Soc **59** (2011), 3–19

[4] F. Wefers and J. Berg. High-performance real-time fir-filtering using fast convolution on graphics hardware. In 13th International Conference on Digital Audio Effects (DAFx-10). Graz, Austria, 2010

[5] D. V. Nikolov, M. J. Mišić, and M. V. Tomašević. Gpu-based implementation of reverb effect. In 2015 23rd Telecommunications Forum Telfor (TELFOR). 2015 990–993

[6] Michael Lester and Jon Boley. The effects of latency on live sound monitoring. In Audio Engineering Society Convention 123. 2007

[7] A. Primavera, S. Cecchi, F. Piazza, J. Li, and Y. Yan. Hybrid reverberator using multiple impulse responses for audio rendering improvement. In 2013 Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing. 2013 314–317

[8] N. Jillings, J. D. Reiss, and R. Stables. Zero-delay large signal convolution using multiple processor architectures. In 2017 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA). ISSN 1947-1629, 2017 339–343

[9] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2018

[10] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. OpenCL programming guide. Pearson Education, 2011

[11] John L Hennessy and David A Patterson. Computer architecture: a quantitative approach. Elsevier, 2011

[12] Nvidia Corporation. OpenCL Programming Guide for the CUDA Architecture, 2009

[13] Advanced Micro Devices Corporation. OpenCL User Guide, 2015